

# Bloom Filter

Jan Werren, Xeno Suter

Dezember 2024

## Inhaltsverzeichnis

<b>1</b>	<b>Was ist ein Bloom Filter und die Idee dahinter</b>	<b>2</b>
<b>2</b>	<b>Vor- und Nachteile von Bloom Filtern</b>	<b>2</b>
2.1	Vorteile . . . . .	2
2.2	Nachteile . . . . .	2
<b>3</b>	<b>Implementierung Bloom Filters</b>	<b>2</b>
3.1	Struktur . . . . .	2
<b>4</b>	<b>Fehlerwahrscheinlichkeit</b>	<b>3</b>
4.1	Ergebnisse der Fehlerwahrscheinlichkeit . . . . .	3
<b>5</b>	<b>Beispiel aus der Praxis</b>	<b>4</b>
<b>6</b>	<b>Programmausgabe</b>	<b>4</b>

# 1 Was ist ein Bloom Filter und die Idee dahinter

Ein Bloom-Filter ist eine probabilistische Datenstruktur, die überprüft, ob ein Element in einem Set enthalten ist. Er basiert auf einem BitSet und mehreren Hash-Funktionen (murmur3-128). Neue Elemente werden durch Setzen von Bits im Array hinzugefügt. Um zu testen, ob ein Element enthalten ist, wird überprüft, ob alle relevanten Bits der Hashes auf 1 gesetzt sind.

## 2 Vor- und Nachteile von Bloom Filtern

### 2.1 Vorteile

- Sehr speichereffizient im Vergleich zu traditionellen Datenstrukturen wie Sets. Speicherplatzkomplexität:  $O(m)$  (wobei  $m$  die Anzahl der Bits im Bloom-Filter ist).
- Schnelle Einfüge- und Prüfoperationen (Zeitkomplexität:  $O(k)$ , wobei  $k$  die Anzahl der Hash-Funktionen ist).
- Keine *false negatives* möglich. Ein Element, das nicht im Set ist, wird immer korrekt als nicht enthalten erkannt.

### 2.2 Nachteile

- *False positives* möglich. Ein Element, das nicht im Set ist, kann fälschlicherweise als enthalten erkannt werden. Die Wahrscheinlichkeit für ein *false positive* steigt mit der Anzahl der Elemente im Set und der Anzahl der Hash-Funktionen.
- Keine Möglichkeit, die Anzahl der Elemente im Set zu bestimmen.
- Keine Entfernung von Elementen möglich (außer mit *Counting Bloom Filters*).

## 3 Implementierung Bloom Filters

### 3.1 Struktur

Ein Bloom-Filter besteht aus einem Bit-Array der Grösse  $m$  und  $k$  Hash-Funktionen. Die Hash-Funktionen werden verwendet, um die Positionen im

Bit-Array zu berechnen, an denen die Bits gesetzt werden. Die Anzahl der Hash-Funktionen beeinflusst die Wahrscheinlichkeit von *false positives*.

## 4 Fehlerwahrscheinlichkeit

Die Fehlerwahrscheinlichkeit wurde wie folgt getestet:

- Eine definierte Anzahl von Wörtern aus einer Datei `words.txt` wurde in den Bloom-Filter eingefügt.
- Anschließend wurden 100'000 zufällige Wörter generiert, die sicher nicht in `words.txt` enthalten waren.
- Für jedes zufällige Wort wurde überprüft, ob der Bloom-Filter es als enthalten markiert.
- Die experimentelle Fehlerwahrscheinlichkeit wurde als Verhältnis der *false positives* zur Anzahl der getesteten Wörter berechnet.
- Des Weiteren wurde die theoretische Fehlerwahrscheinlichkeit berechnet. Mit folgender Formel:

$$P_{\text{false}}(m, n) = \left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right)^k$$

wobei

$$k = \left\lceil \frac{m}{n} \ln(2) \right\rceil$$

### 4.1 Ergebnisse der Fehlerwahrscheinlichkeit

Die berechnete Fehlerwahrscheinlichkeit stimmt eng mit der theoretischen Wahrscheinlichkeit überein. Für  $n = 1000$  Elemente und  $p = 0.01$  ergab der Test:

- Berechnete Fehlerwahrscheinlichkeit: 0.01
- Experimentelle Fehlerwahrscheinlichkeit: 0.0101

## 5 Beispiel aus der Praxis

Ein typisches Anwendungsbeispiel für Bloom-Filter ist in verteilten Systemen wie **Apache Cassandra**. Cassandra verwendet Bloom-Filter, um schnell zu überprüfen, ob ein bestimmter Schlüssel in einer Datenbank vorhanden sein könnte, bevor eine ressourcenintensive Abfrage durchgeführt wird. Dadurch wird die Abfragezeit erheblich reduziert.

## 6 Programmausgabe

```
Bitte geben Sie die Wahrscheinlichkeit ]0..1] an:
0
Die Wahrscheinlichkeit muss grösser als 0 sein! Bitte genau lesen...
Bitte geben Sie die Wahrscheinlichkeit ]0..1] an:
0.04
p: 0.04
n: 58110
m: 389318
k: 5
Experimentelle (generierte Testwörter) Fehlerwahrscheinlichkeit: 0.0392
Berechnete Fehlerwahrscheinlichkeit: 0.04022121808446685
```

Abbildung 1: Screenshot der Programmausgabe